MPP, ShmemPP: Parallel Processing for Sceptics

V. Balaji
SGI/GFDL

29 July 1998

In light of yesterday's fiasco, I thought I'd begin with Leslie Lamport's definition of a distributed system:

A distributed system is one where your program crashes because somewhere something happened to a computer that you never even knew existed.

Overview

- Different hardware models of parallelism: vector, PVP, MPP, DSM

- Programming models for different architectures

- T3E examples of message passing programs

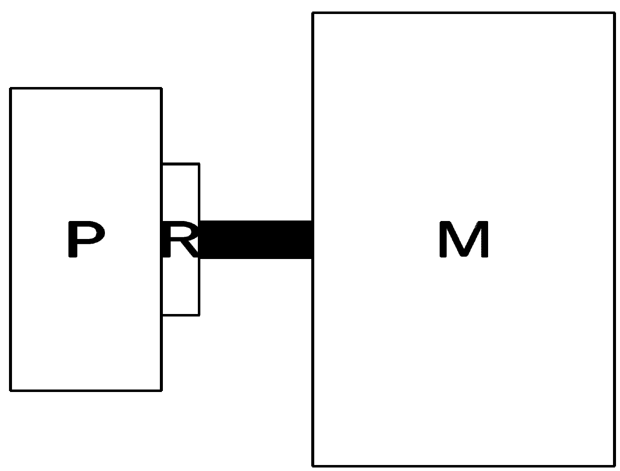
- Why T3E programming is good for you

- Further reading

Sequential computing

The von Neumann model of computing conceptualizes the computer as consisting of a memory where instructions and data are stored, and a processing unit where the computation takes place. At each turn, we fetch an operator and its operands from memory, perform the computation, and write the results back to memory.

```
a = b + c
```



The speed of the computation is constrained by hardware limits: the rate at which instructions and operands can be loaded from memory, and results written back; and the speed of the processing units. The overall computation rate is limited by the slower of the two: memory.

Latency: time to find a word.

Bandwidth: number of words per unit time that can stream through the pipe.

A processor clock period is currently $\sim$ 2-4 ns, Moore's constant is $4\times/3$ years.

DRAM latency is $\sim$ 60 ns, Moore's constant is $1.3\times/3$ years.

Maximum memory bandwidth is theoretically the same as the clock speed, but far less for commodity memory.

Furthermore, since memory and processors are built basically of the same "stuff", there is no way to reverse this trend.

Within the raw physical limitations on processor and memory, there are algorithmic and architectural ways to speed up computation. Most involve doing more than one thing at once.

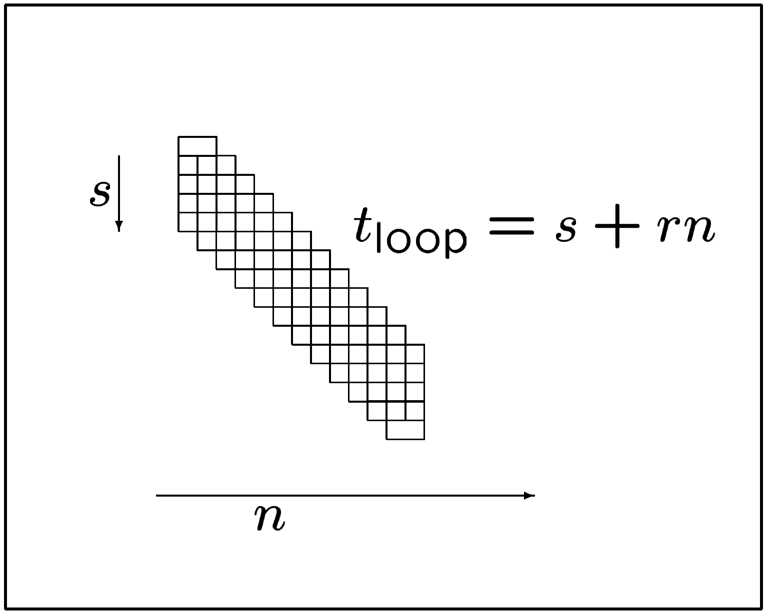
- Overlap separate computations and/or memory operations.
  - Pipelining.
  - Multiple functional units.
  - Overlap computation with memory operations.
  - Re-use already fetched information: **caching**.
  - Memory pipelining.
- Multiple computers sharing data.

The search for **concurrency** becomes a major element in the design of algorithms (and libraries, and compilers). Concurrency can be sought at different grain sizes.

Vector computing

Cray: if the same operation is *independently* performed on many different operands, schedule the operands to stream through the processing unit at a rate $r = 1$ per CP. Thus was born **vector processing**.

```
do i = 1,n
   a(i) = b(i) + c(i)
enddo
```



So long as the computations for each instance of the loop can be concurrently scheduled, the work within the loop can be made as complicated as one wishes.

The magic of vector computing is that for $s \gg rn$, $t_{\text{loop}} \approx s$ for any length n!

Of course in practice s depends on n if we consider the cost of fetching n operands from memory and loading the vector registers.

Vector machines tend to be expensive since they must use the fastest memory technology available to use the full potential of vector pipelining.

Real codes in general cannot be recast as a single loop of n concurrent sequences of arithmetic operations. There is lots of other stuff to be done (memory management, I/O, etc.) Since sustained memory bandwidth requirements over an entire code are somewhat lower, we can let multiple processors share the bandwidth, and seek concurrency at a coarser grain size.

```
!mic$ DOALL private(j)
do j = 1,n
   call ocean(j)
   call atmos(j)
enddo
```

Since the language standards do not specify parallel constructs, they are inserted through compiler directives.

Amdahl's Law

Even a well-parallelized code will have some serial work, such as initialization, I/O operations, etc. The time to execute a parallel code on P processors is given by

$$t_P = t_s + \frac{t_{\|}}{P} \tag{1}$$

$$\frac{t_1}{t_P} = \frac{1}{s + \frac{1-s}{P}} \tag{2}$$

where $s \equiv \frac{t_s}{t_1}$ is the serial fraction.

Speedup of a 5% serial code is at most 20.

Load-balancing

If the computational cost per instance of a parallel region unequal, the loop as a whole executes at the speed of the slowest instance (implicit synchronization at the end of a parallel region).

Work must be partitioned in a way that keeps the load on each parallel leg roughly equal.

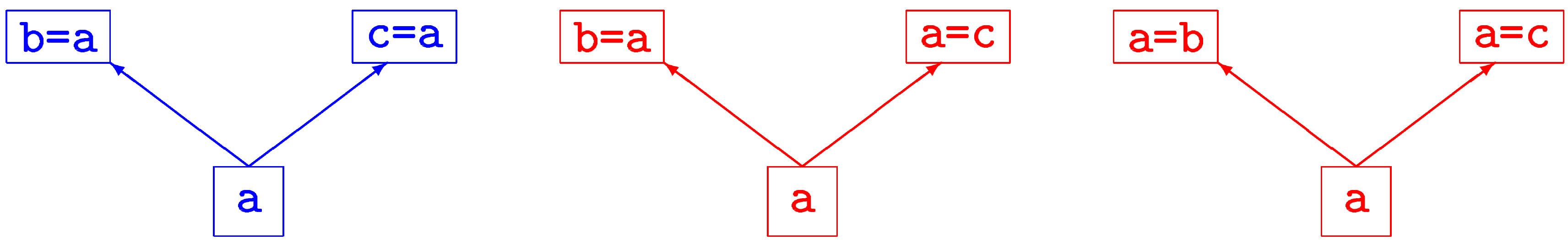
If there is sufficient granularity (several instances of a parallel loop per processor), this can be automatically accomplished by implementing a global task queue.

Memory model for shared memory parallelism

Every variable has a **scope**: global or local. Writing to global variables in a parallel region can result in a **race condition.**

Text	Static data	Shared stack	Shared heap	Local stack	Local heap

Race conditions



The second and third case result in a race condition and unpredictable results. The third case may be OK for certain reduction or search operations, defined within a **critical region.**

```
!mic$ GUARD
a = a + b
!mic$ END GUARD
```

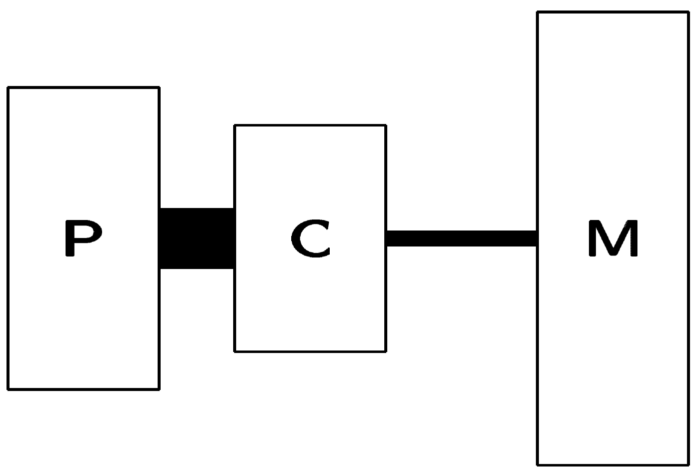
Shared memory parallelism with a **flat** or **uniform** memory model does not **scale** to large numbers of processors, because (again) of memory bandwidth. UMA memory access quickly runs out of aggregate bandwidth.

Scalability: the number of processors you can usefully add to a parallel system. It is also used to describe something like the degree of coarse-grained concurrency in a code or an algorithm, but this use is somewhat suspect, as this is almost always a function of problem size.

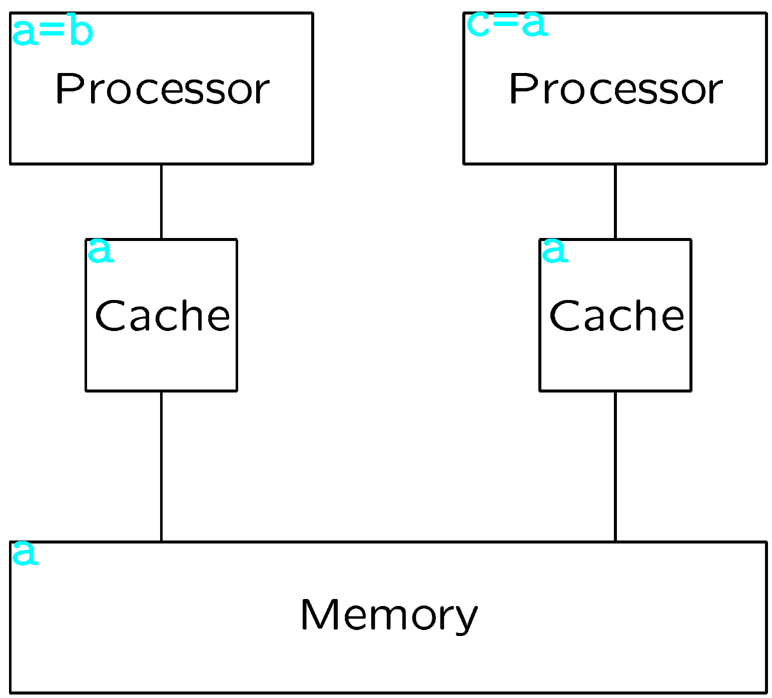
Caches

The memory bandwidth bottleneck may be alleviated by the use of caches.

Caches exploit **temporal locality** of memory access requests. Memory latency is also somewhat obscured by exploiting **spatial locality** as well: when a word is requested, adjacent words, constituting a **cache line**, are fetched as well.



In a multi-processor environment, extra overhead is incurred to maintain **cache coherency**.



To summarize:

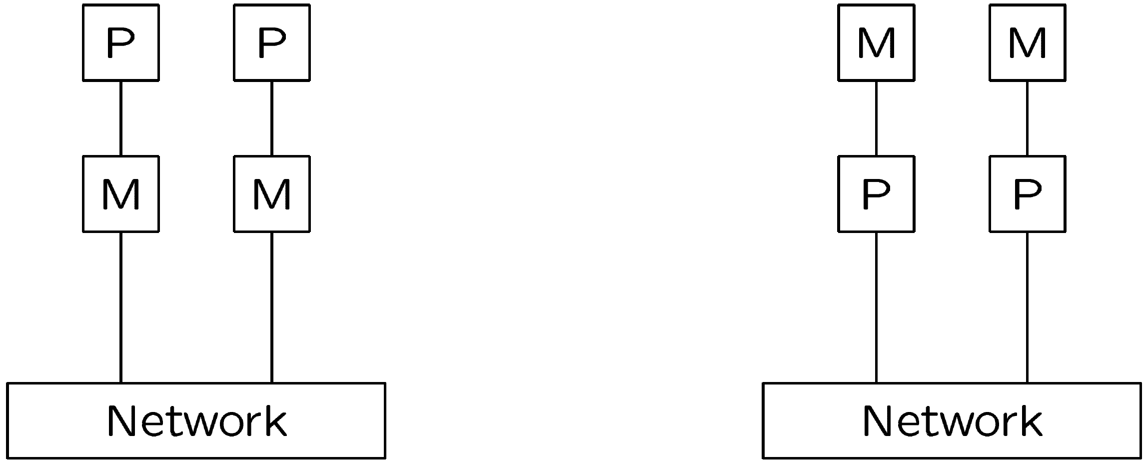
UMA architectures suffer from a crisis of aggregate memory bandwidth. The use of caches may alleviate the bandwidth problem, but require some form of communication between the disjoint members of the system: processors or caches.

This suggests dispensing with the UMA model altogether: moving toward a model where memory segments are themselves distributed and communicate over a network. This involves a radical change to the programming model, since there is no longer a single **address space** in it. Instead communication between disjoint regions must be explicit: **message passing.**

More recently, with the advent of fast cache-coherency techniques, the single-address-space programming model has been revived within the **ccNUMA** architectural model. Here memory is *physically* distributed, but *logically* shared. More on this later if we have time.

Distributed Memory Systems: MPP

Processing elements (PEs), consisting of a processor *and* memory, are distributed across a network, and exchange data only as required, by explicit send and receive.

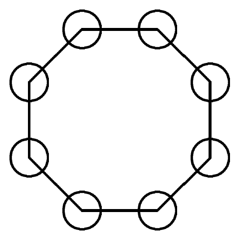


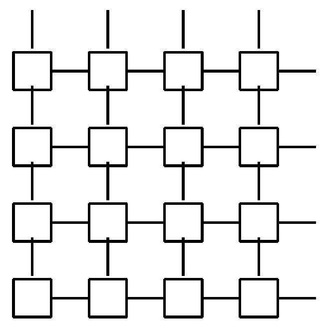
Tightly coupled systems: memory closer to network than processor.
Loosely coupled systems: processor closer to network than memory.

Loose coupling could include heterogeneous computing across a LAN/WAN/Internet.

Network Topologies

Ring, hypercube, torus.





A torus provides scalable connectivity: an n-dimensional torus of side p has p^n PEs with a maximum distance of $\frac{pn}{2}$.

T3E: the canonical MPP supercomputer

The T3E is a tightly-coupled distributed memory supercomputer with a 3D-torus low-latency high-speed interconnect, scalable to a maximum of 2048 processors.

The memory hierarchy includes two levels of cache.

The processor is among the fastest available *commodity* processors: the Dec Alpha (600 MHz clock latest). There are two arithmetic processing units: allowing us to call it a T3E-1200.

A parallel job is granted p physically contiguous PEs when available. The job runs continously on these until done (no time-sharing, swapping, etc.)

Your first T3E program

```
program test
print *, 'PE', my_pe(), ' says hello.'
if( my_pe().eq.0 )print *, 'Total number of PEs is', num_pes()
end

t3e 3% f90 -Wl"-X m" a.f90
t3e 4% mpprun -n4 a.out
 PE 2  says hello.
 PE 3  says hello.
 PE 1  says hello.
 PE 0  says hello.
 Total number of PEs is 4
```

Message passing

```
program test
integer :: right
right = my_pe()
call BARRIER()
call SHMEM_GET( right, right, 1, mod(my_pe()+1,num_pes()) )
print *, 'PE', my_pe(), ' says hi to its neighbour on the right,', right
end

t3e 6% f90 -Wl"-X m" a.f90
t3e 7% mpprun -n4 a.out
 PE 2  says hi to its neighbour on the right, 3
 PE 0  says hi to its neighbour on the right, 1
 PE 3  says hi to its neighbour on the right, 0
 PE 1  says hi to its neighbour on the right, 2
```

Communication and synchronization

The SHMEM library is written with a tightly-coupled MPP like T3E in mind, where memory is close to the network. This permits the PE wishing to `get()` or `put()` data to a remote PE to proceed without interrupting the remote PE.

Requires a synchronization operation to make sure the transmitted data is available for the operation. Synchronization is effected with a `barrier()` call. On loosely-coupled systems barriers can be very expensive, on T3E it is implemented in the hardware and is extremely fast.

`get()` synchronization:

```
a = ...
call BARRIER()
call SHMEM_GET( a, a, 1, remote_PE )
...
```

`put()` synchronization:

```
call SHMEM_PUT( a, a, 1, remote_PE )
...
call BARRIER()
a = ...
```

`put()` returns control to the sender after initiating communication. `get()` is a blocking operation.

MPI: a communication model for loosely-coupled systems

For a loosely-coupled or heterogeneous system, direct operations to a remote memory cannot be permitted.

The communication model is a **rendezvous**.

```
call MPI_SEND( a, ..., to_pe,   ... )
call MPI_RECV( b, ..., from_pe, ... )
```

There is now another level of latency – **software latency** – in negotiating the communication.

Besides `put()`s and `get()`s, the SHMEM library contains several collective and global operations:

```
SHMEM_BROADCAST(), SHMEM_COLLECT() collective operations
SHMEM_SUM(), SHMEM_MAX(), SHMEM_AND(), ... global reductions
SHMEM_SWAP(), SHMEM_INC(), ... remote atomic operations
SHMEM_IXPUT(), SHMEM_IXGET(), ... strided gets and puts
```

Collective and global operations can be performed on a subset of PEs as well.

Implementing a global sum

Sum the value of `a` on all PEs, every PE to have a copy of the result. Simplest algorithm: gather on PE 0, sum and broadcast.

```
program test
real :: a, sum
a = my_pe()
call BARRIER()
if( my_pe().EQ.0 )then
    sum = a
    do n = 1,num_pes()-1
       call SHMEM_GET( a, a, 1, n )
       sum = sum + a
    enddo
    do n = 1,num_pes()-1
       call SHMEM_PUT( sum, sum, 1, n )
    enddo
endif
call BARRIER()
print *, 'sum=', sum, ' on PE', my_pe()
end
```

```
t3e 25% f90 -Wl"-X m" ~/src/examples/sum_on_PE0.f90
t3e 26% mpprun -n4 a.out
 sum= 6.  on PE 0
 sum= 6.  on PE 1
 sum= 6.  on PE 2
 sum= 6.  on PE 3
```

This algorithm on p processors involves $2p$ communications and p summations, all sequential.

Here's another algorithm for doing the same thing: a binary tree. It executes in $\log_2 p$ steps, each step consisting of one communication and one summation.

$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$	$\boxed{5}$	$\boxed{6}$	$\boxed{7}$	$\boxed{8}$
$\boxed{\sum_1^2}$	$\boxed{\sum_1^2}$	$\boxed{\sum_3^4}$	$\boxed{\sum_3^4}$	$\boxed{\sum_5^6}$	$\boxed{\sum_5^6}$	$\boxed{\sum_7^8}$	$\boxed{\sum_7^8}$
$\boxed{\sum_1^4}$	$\boxed{\sum_1^4}$	$\boxed{\sum_1^4}$	$\boxed{\sum_1^4}$	$\boxed{\sum_5^8}$	$\boxed{\sum_5^8}$	$\boxed{\sum_5^8}$	$\boxed{\sum_5^8}$
$\boxed{\sum_1^8}$	$\boxed{\sum_1^8}$	$\boxed{\sum_1^8}$	$\boxed{\sum_1^8}$	$\boxed{\sum_1^8}$	$\boxed{\sum_1^8}$	$\boxed{\sum_1^8}$	$\boxed{\sum_1^8}$

There are two ways to perform each step:

```
if( mod(pe,2).EQ.0 )then !execute on even-numbered PEs
    call SHMEM_GET( a, sum, 1, pe+1 )
    sum = sum + a
    call SHMEM_PUT( sum, sum, 1, pe+1 )
endif

if( mod(pe,2).EQ.0 )then !execute on even-numbered PEs
    call SHMEM_GET( a, sum, 1, pe+1 )
    sum = sum + a
else                     !execute on odd-numbered PEs
    call SHMEM_GET( a, sum, 1, pe-1 )
    sum = sum + a
endif
```

The second is faster, even though a redundant computation is performed.

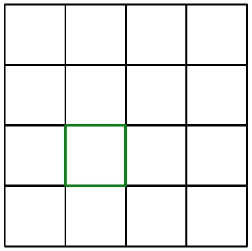
SHMEMs in general require contiguous data or a regular stride.

```
dimension a(400,400), b(100,100), c(100,100,16)

call SHMEM_GET( a(101:200,101:200), a(101:200,101:200), 10000, remote_PE )

b = a(101:200,101:200)
call BARRIER()
call SHMEM_GET( b, b, 10000, remote_PE )
a(101:200,101:200) = b

call SHMEM_GET( c(1,1,6), c(1,1,6), 10000, remote_PE )
```



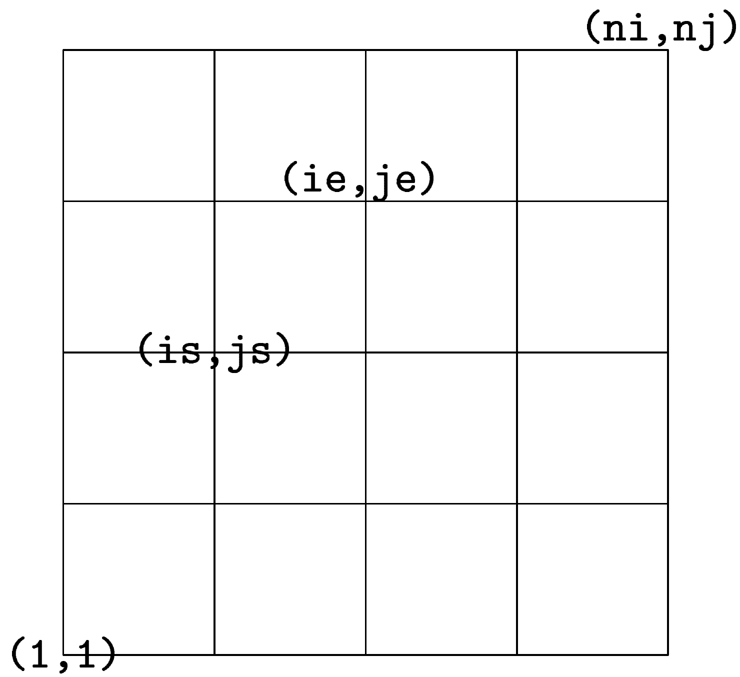
It is best to lay out data in suitable blocks.

Domain decomposition

```
do j = 1,nj
   do i = 1,ni
      a(i,j) = ...
   enddo
enddo
```

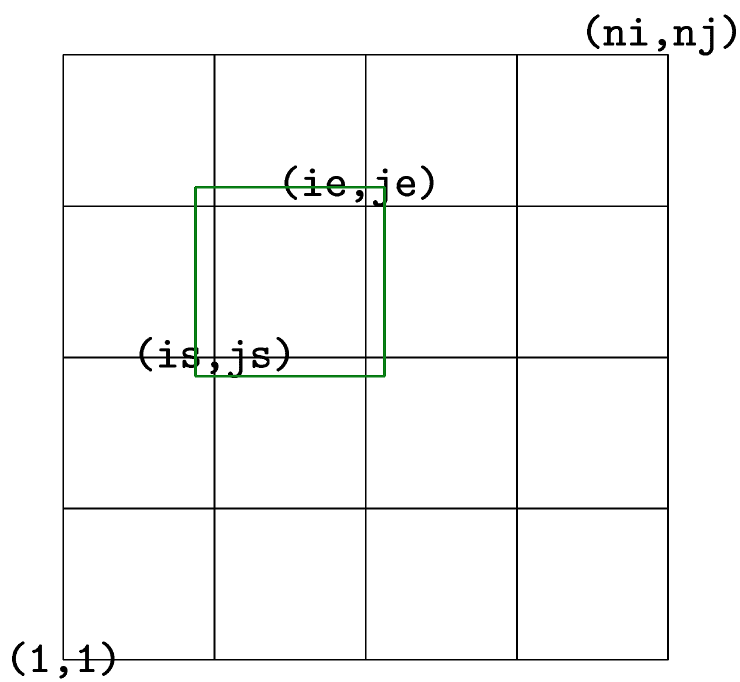
is replaced by

```
do j = js,je
   do i = is,ie
      a(i,j) = ...
   enddo
enddo
```



Computational and data domains

```
do j = js,je
   do i = is,ie
         a(i,j) = ...  + a(i-1,j+1) + ...
   enddo
enddo
```



The **computational domain** is the set of gridpoints that are computed on a domain. The **data domain** is the set of gridpoints needs to be available on-processor to carry out the computation.

The data domain may consist of a **halo** of a certain width, or it might be global along an axis (e.g polar filter).

There is a f90 module available for those who are interested, that has simple domain decomposition and communication interfaces. It will provide a uniform interface to SHMEM and MPI calls for communication. (MPI version is not quite ready.)

```
call mpp_define_domains(...)
call mpp_update_domains(...)
call mpp_transmit(...)
```

The overall model structure can be the same for both shared memory and message passing codes, though algorithms may have to change at some other (smaller) grain size.

```
 call mpp_define_domains(...)
...
!mic$ DOALL private(j)
do j = 1,n
   if( domain(j)%pe.NE.my_pe() )cycle
   call ocean(j)
   call atmos(j)
enddo
...
call mpp_update_domains(...)
```

Parallel I/O

"I/O certainly has been lagging in the last decade." – Seymour Cray, Public Lecture (1976).

"Also, I/O needs a lot of work." – David Kuck, Keynote Address, 15th Annual Symposium on Computer Architecture (1988).

"I/O has been the orphan of computer architecture." – Hennessy and Patterson, Computer Architecture - A Quantitative Approach. 2nd Ed. (1996).

The `global` FFIO layer for parallel I/O for the T3E appears to be a correct formulation of the issue, but still needs work.

Why T3E programming is good for you

- T3E is the canonical MPP. The programming model is textbook-clean, and needs to be understood before more complicated models are attempted.

- With a superlative interconnect, and hardware synchronization, it is an ideal machine to cut your message-passing teeth on. Many techniques of "defensive programming" for slower networks and communication models can be avoided.

- Nice tools: Apprentice, Totalview.

- Nice on-site analysts.

Furthermore...

- The hierarchical memory model is inescapable for the foreseeable future. Algorithms for this memory model are in gross and in subtle ways different from flat-memory algorithms.

- **Algorithmic granularity** is important to consider. Overall structures can remain the same between shared-memory and message-passing models. At some intermediate grain size, models have to be recast for hierarchical memory. While tuning toward specific hierarchies can appear to be a daunting task, at an appropriate grain size, machine specifics can be left to libraries and compilers.

- With the advent of DSM and ccNUMA, it is now a well-posed problem whether shared-memory or message-passing is the better programming model.

Bibliography

- Designing and Building Parallel Programs: an online textbook by Ian Foster. `http://www.mcs.anl.gov/dbpp/text/book.html`

- Highly Parallel Computing, Almasi and Gottlieb.

- Parallel Computer Architecture: A Hardware/Software Approach, Culler and Singh.

- In search of clusters, Pfister.

- Solving Problems on Concurrent Processors, vol 1.: General Techniques and Regular Problems, Fox et al.

- Cray manuals!